This Page Blank (uspto)

PCT

WORLD INTELLECTUAL PROPERTY ORGANIZATION
International Bureau

INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification 7 : G06F 9/38 A2

(11) International Publication Number: WO 00/33185

(43) International Publication Date: 8 June 2000 (08.06.00)

(21) International Application Number: PCT/US99/28821

(22) International Filing Date: 3 December 1999 (03.12.99)

(30) Priority Data:
09/204,480 3 December 1998 (03.12.98) US

(71) Applicant: SUN MICROSYSTEMS, INC. [US/US]; 901 San Antonio Road, M/S PAL01-521, Palo Alto, CA 94303 (US).

(72) Inventors: TREMBLAY, Marc; 140 Hanna Way, Menlo Park, CA 94025 (US). JOY, William, N.; P.O. Box 23, Aspen, CO 81612-0023 (US).

(74) Agents: KOESTNER, Ken, J. et al.; Skjerven, Morrill, MacPherson, Franklin & Friel LLP, Suite 700, 25 Metro Drive, San Jose, CA 95110 (US).

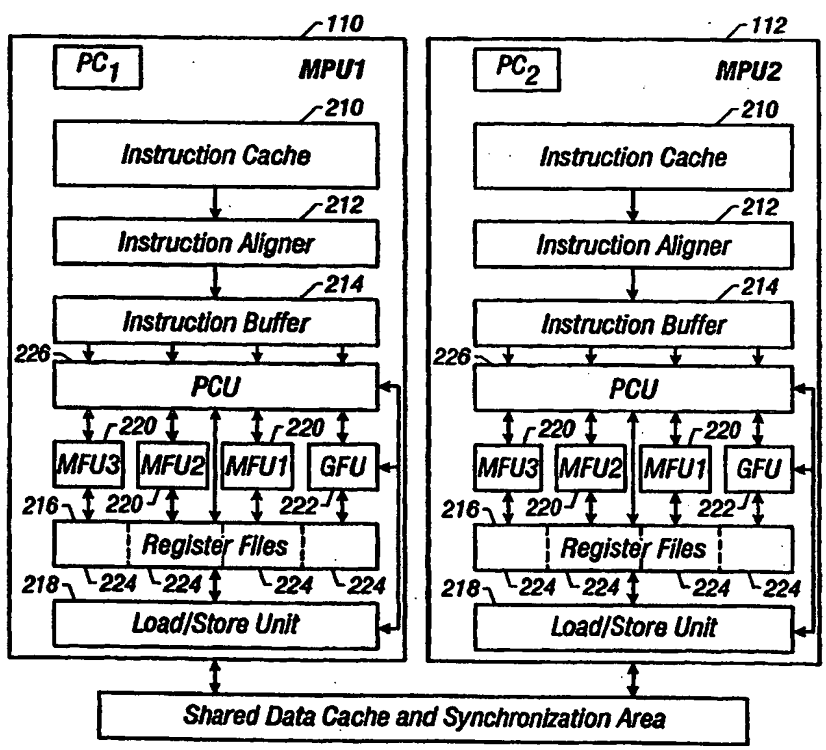
(81) Designated States: JP, KR, European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE).

Published
Without international search report and to be republished upon receipt of that report.

(54) Title: A MULTIPLE-THREAD PROCESSOR FOR THREADED SOFTWARE APPLICATIONS

(57) Abstract

A processor has an improved architecture for multiple-thread operation on the basis of a highly parallel structure including multiple independent parallel execution paths for executing in parallel across threads and a multiple-instruction parallel pathway within a thread. The multiple independent parallel execution paths include functional units that execute an instruction set including special data-handling instructions that are advantageous in a multiple-thread environment.



FOR THE PURPOSES OF INFORMATION ONLY

Codes used to identify States party to the PCT on the front pages of pamphlets publishing international applications under the PCT.

AL	Albania	ES	Spain	LS	Lesotho	SI	Slovenia
AM	Armenia	FI	Finland	LT	Lithuania	SK	Slovakia
AT	Austria	FR	France	LU	Luxembourg	SN	Senegal
AU	Australia	GA	Gabon	LV	Latvia	SZ	Swaziland
AZ	Azerbaijan	GB	United Kingdom	MC	Monaco	TD	Chad
BA	Bosnia and Herzegovina	GE	Georgia	MD	Republic of Moldova	TG	Togo
BB	Barbados	GH	Ghana	MG	Madagascar	TJ	Tajikistan
BE	Belgium	GN	Guinea	MK	The former Yugoslav Republic of Macedonia	TM	Turkmenistan
BF	Burkina Faso	GR	Greece			TR	Turkey
BG	Bulgaria	HU	Hungary	ML	Mali	TT	Trinidad and Tobago
BJ	Benin	IE	Ireland	MN	Mongolia	UA	Ukraine
BR	Brazil	IL	Israel	MR	Mauritania	UG	Uganda
BY	Belarus	IS	Iceland	MW	Malawi	US	United States of America
CA	Canada	IT	Italy	MX	Mexico	UZ	Uzbekistan
CF	Central African Republic	JP	Japan	NE	Niger	VN	Viet Nam
CG	Congo	KE	Kenya	NL	Netherlands	YU	Yugoslavia
CH	Switzerland	KG	Kyrgyzstan	NO	Norway	ZW	Zimbabwe
CI	Côte d'Ivoire	KP	Democratic People's Republic of Korea	NZ	New Zealand		
CM	Cameroon			PL	Poland		
CN	China	KR	Republic of Korea	PT	Portugal		
CU	Cuba	KZ	Kazakstan	RO	Romania		
CZ	Czech Republic	LC	Saint Lucia	RU	Russian Federation		
DE	Germany	LI	Liechtenstein	SD	Sudan		
DK	Denmark	LK	Sri Lanka	SE	Sweden		
EE	Estonia	LR	Liberia	SG	Singapore		

A MULTIPLE-THREAD PROCESSOR FOR THREADED SOFTWARE APPLICATIONS

TECHNICAL FIELD

The present invention relates to a processor architecture. More specifically, the present invention relates to a single-chip processor architecture including structures for multiple-thread operation.

BACKGROUND ART

For various processing applications, an automated system may handle multiple events or processes concurrently. A single process is termed a thread of control, or "thread", and is the basic unit of operation of independent dynamic action within the system. A program has at least one thread. A system performing concurrent operations typically has many threads, some of which are transitory and others enduring. Systems that execute among multiple processors allow for true concurrent threads. Single-processor systems can only have illusory concurrent threads, typically attained by time-slicing of processor execution, shared among a plurality of threads.

Some programming languages are particularly designed to support multiple-threading. One such language is the Java™ programming language that is advantageously executed using an abstract computing machine, the Java Virtual Machine™. A Java Virtual Machine™ is capable of supporting multiple threads of execution at one time. The multiple threads independently execute Java code that operates on Java values and objects residing in a shared main memory. The multiple threads may be supported using multiple hardware processors, by time-slicing a single hardware processor, or by time-slicing many hardware processors. in 1990 programmers at Sun Microsystems developed a universal programming language, eventually known as "the Java™ programming language". Java™, Sun, Sun Microsystems and the Sun Logo are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries. All SPARC trademarks, including UltraSPARC I and UltraSPARC II, are used under license and are trademarks of SPARC International, Inc. in the United States and other countries. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc.

Java™ supports the coding of programs that, though concurrent, exhibit deterministic behavior, by including techniques and structures for synchronizing the concurrent activity of threads. To synchronize threads, Java™ uses monitors, high-level constructs that allow only a single thread at one time to execute a region of code protected by the monitor. Monitors use locks associated with executable objects to control thread execution.

A thread executes code by performing a sequence of actions. A thread may use the value of a variable or assign the variable a new value. If two or more concurrent threads act on a shared variable, the actions on the variable may produce a timing-dependent result, an inherent consequence of concurrent programming.

Each thread has a working memory that may store copies of the values of master copies of variables from main memory that are shared among all threads. A thread usually accesses a shared variable by obtaining a lock and flushing the working memory of the thread, guaranteeing that shared values are thereafter loaded from the shared memory to the working memory of the thread. By unlocking a lock, a thread guarantees that the values held by the thread in the working memory are written back to the main memory.

Several rules of execution order constrain the order in which certain events may occur. For example, actions performed by one thread are totally ordered so that for any two actions performed by a thread, one action precedes the other. Actions performed by the main memory for any one variable are totally ordered so that for any two actions performed by the main memory on the same variable, one action precedes the other. Actions performed by the main memory for any one lock are totally ordered so that for any two actions performed by the main memory on the same lock, one action precedes the other. Also, an action is not permitted to follow itself. Threads do not interact directly but rather only communicate through the shared main memory.

The relationships among the actions of a thread and the actions of main memory are also constrained by rules. For example, each lock or unlock is performed jointly by some thread and the main memory. Each load action by a thread is uniquely paired with a read action by the main memory such that the load action follows the read action. Each store action by a thread is uniquely paired with a write action by the main memory such that the write action follows the store action.

An implementation of threading incurs some overhead. For example, a single processor system incurs overhead in time-slicing between threads. Additional overhead is incurred in allocating and handling accessing of main memory and local thread working memory.

What is needed is a processor architecture that supports multiple-thread operation and reduces the overhead associated with multiple-thread operation.

DISCLOSURE OF INVENTION

A processor has an improved architecture for multiple-thread operation on the basis of a highly parallel structure including multiple independent parallel execution paths for executing in parallel across threads and a multiple-instruction parallel pathway within a thread. The multiple independent parallel execution paths include functional units that execute an instruction set including special data-handling instructions that are advantageous in a multiple-thread environment.

In accordance with one embodiment of the present invention, a general-purpose processor includes two independent processor elements in a single integrated circuit die. The dual independent processor elements advantageously execute two independent threads concurrently during multiple-threading operation. When only a single thread is executed on a first of the two processor elements, the second processor element is

advantageously used to perform garbage collection, Just-In-Time (JIT) compilation, and the like. Illustratively, the independent processor elements are Very Long Instruction Word (VLIW) processors. For example, one illustrative processor includes two independent Very Long Instruction Word (VLIW) processor elements, each of which executes an instruction group or instruction packet that includes up to four instructions, otherwise termed subinstructions. Each of the instructions in an instruction group executes on a separate functional unit.

The two threads execute independently on the respective VLIW processor elements, each of which includes a plurality of powerful functional units that execute in parallel. In the illustrative embodiment, the VLIW processor elements have four functional units including three media functional units and one general functional unit. All of the illustrative media functional units include an instruction that executes both a multiply and an add in a single cycle, either floating point or fixed point.

In accordance with an aspect of the present invention, an individual independent parallel execution path has operational units including instruction supply blocks and instruction preparation blocks, functional units, and a register file that are separate and independent from the operational units of other paths of the multiple independent parallel execution paths. The instruction supply blocks include a separate instruction cache for the individual independent parallel execution paths, however the multiple independent parallel execution paths share a single data cache since multiple threads sometimes share data. The data cache is dual-ported, allowing data access in both execution paths in a single cycle.

In addition to the instruction cache, the instruction supply blocks in an execution path include an instruction aligner, and an instruction buffer that precisely format and align the full instruction group to prepare to access the register file. An individual execution path has a single register file that is physically split into multiple register file segments, each of which is associated with a particular functional unit of the multiple functional units. At any point in time, the register file segments as allocated to each functional unit each contain the same content. A multi-ported register file is typically metal limited to the area consumed by the circuit proportional with the square of the number of ports. It has been discovered that a processor having a register file structure divided into a plurality of separate and independent register files forms a layout structure with an improved layout efficiency. The read ports of the total register file structure are allocated among the separate and individual register files. Each of the separate and individual register files has write ports that correspond to the total number of write ports in the total register file structure. Writes are fully broadcast so that all of the separate and individual register files are coherent.

BRIEF DESCRIPTION OF DRAWINGS

The features of the described embodiments are specifically set forth in the appended claims. However, embodiments of the invention relating to both structure and method of operation, may best be understood by referring to the following description and accompanying drawings.

FIGURE 1 is a schematic block diagram illustrating a single integrated circuit chip implementation of a processor in accordance with an embodiment of the present invention.

FIGURE 2 is a schematic block diagram showing the core of the processor.

FIGURE 3 is a schematic block diagram that illustrates an embodiment of the split register file that is suitable for usage in the processor.

FIGURE 4 is a schematic block diagram that shows a logical view of the register file and functional units in the processor.

FIGURE 5 is a pictorial schematic diagram depicting an example of instruction execution among a plurality of media functional units.

FIGURE 6 illustrates a schematic block diagram of an SRAM array used for the multi-port split register file.

FIGURE 7A and **7B** are, respectively, a schematic block diagram and a pictorial diagram that illustrate the register file and a memory array insert of the register file.

FIGURE 8 is a schematic block diagram showing an arrangement of the register file into the four register file segments.

FIGURE 9 is a schematic timing diagram that illustrates timing of the processor pipeline.

The use of the same reference symbols in different drawings indicates similar or identical items.

MODES FOR CARRYING OUT THE INVENTION

Referring to **FIGURE 1**, a schematic block diagram illustrates a processor **100** having an improved architecture for multiple-thread operation on the basis of a highly parallel structure including multiple independent parallel execution paths, shown herein as two media processing units **110** and **112**. The execution paths execute in parallel across threads and include a multiple-instruction parallel pathway within a thread. The multiple independent parallel execution paths include functional units executing an instruction set having special data-handling instructions that are advantageous in a multiple-thread environment.

The multiple-threading architecture of the processor **100** is advantageous for usage in executing multiple-threaded applications using a language such as the Java™ language running under a multiple-threaded operating system on a multiple-threaded Java Virtual Machine™. The illustrative processor **100** includes two independent processor elements, the media processing units **110** and **112**, forming two independent parallel execution paths. A language that supports multiple threads, such as the Java™ programming language generates two threads that respectively execute in the two parallel execution paths with very little overhead incurred. The special instructions executed by the multiple-threaded processor include instructions for accessing arrays, and instructions that support garbage collection.

A single integrated circuit chip implementation of a processor **100** includes a memory interface **102**, a geometry decompressor **104**, the two media processing units **110** and **112**, a shared data cache **106**, and several interface controllers. The interface controllers support an interactive graphics environment with real-time constraints by integrating fundamental components of memory, graphics, and input/output bridge functionality on a single die. The components are mutually linked and closely linked to the processor core with high bandwidth, low-latency communication channels to manage multiple high-bandwidth data streams efficiently and with a low response time. The interface controllers include a an UltraPort Architecture Interconnect (UPA) controller **116** and a peripheral component interconnect (PCI) controller **120**. The illustrative memory interface **102** is a direct Rambus dynamic RAM (DRDRAM) controller. The shared data cache **106** is a dual-ported storage that is shared among the media processing units **110** and **112** with one port allocated to each media processing unit. The data cache **106** is four-way set associative, follows a write-back protocol, and supports hits in the fill buffer (not shown). The data cache **106** allows fast data sharing and eliminates the need for a complex, error-prone cache coherency protocol between the media processing units **110** and **112**.

The UPA controller **116** is a custom interface that attains a suitable balance between high-performance computational and graphic subsystems. The UPA is a cache-coherent, processor-memory interconnect. The UPA attains several advantageous characteristics including a scaleable bandwidth through support of multiple bused interconnects for data and addresses, packets that are switched for improved bus utilization, higher bandwidth, and precise interrupt processing. The UPA performs low latency memory accesses with high throughput paths to memory. The UPA includes a buffered cross-bar memory interface for increased bandwidth and improved scaleability. The UPA supports high-performance graphics with two-cycle single-word writes on the 64-bit UPA interconnect. The UPA interconnect architecture utilizes point-to-point packet switched messages from a centralized system controller to maintain cache coherence. Packet switching improves bus bandwidth utilization by removing the latencies commonly associated with transaction-based designs.

The PCI controller **120** is used as the primary system I/O interface for connecting standard, high-volume, low-cost peripheral devices, although other standard interfaces may also be used. The PCI bus

effectively transfers data among high bandwidth peripherals and low bandwidth peripherals, such as CD-ROM players, DVD players, and digital cameras.

Two media processing units **110** and **112** are included in a single integrated circuit chip to support an execution environment exploiting thread level parallelism in which two independent threads can execute simultaneously. The threads may arise from any sources such as the same application, different applications, the operating system, or the runtime environment. Parallelism is exploited at the thread level since parallelism is rare beyond four, or even two, instructions per cycle in general purpose code. For example, the illustrative processor **100** is an eight-wide machine with eight execution units for executing instructions. A typical "general-purpose" processing code has an instruction level parallelism of about two so that, on average, most (about six) of the eight execution units would be idle at any time. The illustrative processor **100** employs thread level parallelism and operates on two independent threads, possibly attaining twice the performance of a processor having the same resources and clock rate but utilizing traditional non-thread parallelism.

Thread level parallelism is particularly useful for Java™ applications, which are bound to have multiple threads of execution. Java™ methods including "suspend", "resume", "sleep", and the like include effective support for threaded program code. In addition, Java™ class libraries are thread-safe to promote parallelism. Furthermore, the thread model of the processor **100** supports a dynamic compiler which runs as a separate thread using one media processing unit **110** while the second media processing unit **112** is used by the current application. In the illustrative system, the compiler applies optimizations based on "on-the-fly" profile feedback information while dynamically modifying the executing code to improve execution on each subsequent run. For example, a "garbage collector" may be executed on a first media processing unit **110**, copying objects or gathering pointer information, while the application is executing on the other media processing unit **112**.

Although the processor **100** shown in **FIGURE 1** includes two processing units on an integrated circuit chip, the architecture is highly scaleable so that one to several closely-coupled processors may be formed in a message-based coherent architecture and resident on the same die to process multiple threads of execution. Thus, in the processor **100**, a limitation on the number of processors formed on a single die thus arises from capacity constraints of integrated circuit technology rather than from architectural constraints relating to the interactions and interconnections between processors.

The processor **100** is a general-purpose processor that includes the media processing units **110** and **112**, two independent processor elements in a single integrated circuit die. The dual independent processor elements **110** and **112** advantageously execute two independent threads concurrently during multiple-threading operation. When only a single thread executes on the processor **100**, one of the two processor elements executes the thread, the second processor element is advantageously used to perform garbage collection, Just-In-Time (JIT) compilation, and the like. In the illustrative processor **100**, the independent processor elements **110** and **112** are Very Long Instruction Word (VLIW) processors. For example, one illustrative processor **100**

includes two independent Very Long Instruction Word (VLIW) processor elements, each of which executes an instruction group or instruction packet that includes up to four instructions. Each of the instructions in an instruction group executes on a separate functional unit.

The usage of a VLIW processor advantageously reduces complexity by avoiding usage of various structures such as schedulers or reorder buffers that are used in superscalar machines to handle data dependencies. A VLIW processor typically uses software scheduling and software checking to avoid data conflicts and dependencies, greatly simplifying hardware control circuits.

The two threads execute independently on the respective VLIW processor elements **110** and **112**, each of, which includes a plurality of powerful functional units that execute in parallel. In the illustrative embodiment shown in **FIGURE 2**, the VLIW processor elements **110** and **112** have four functional units including three media functional units **220** and one general functional unit **222**. All of the illustrative media functional units **220** include an instruction that executes both a multiply and an add in a single cycle, either floating point or fixed point. Thus, a processor with two VLIW processor elements can execute twelve floating point operations each cycle. At a 500 MHz execution rate, for example, the processor runs at an 6 gigaflop rate, even without accounting for general functional unit operation.

Referring to **FIGURE 2**, a schematic block diagram shows the core of the processor **100**. The media processing units **110** and **112** each include an instruction cache **210**, an instruction aligner **212**, an instruction buffer **214**, a pipeline control unit **226**, a split register file **216**, a plurality of execution units, and a load/store unit **218**. In the illustrative processor **100**, the media processing units **110** and **112** use a plurality of execution units for executing instructions. The execution units for a media processing unit **110** include three media functional units (MFU) **220** and one general functional unit (GFU) **222**.

An individual independent parallel execution path **110** or **112** has operational units including instruction supply blocks and instruction preparation blocks, functional units **220** and **222**, and a register file **216** that are separate and independent from the operational units of other paths of the multiple independent parallel execution paths. The instruction supply blocks include a separate instruction cache **210** for the individual independent parallel execution paths, however the multiple independent parallel execution paths share a single data cache **106** since multiple threads sometimes share data. The data cache **106** is dual-ported, allowing data access in both execution paths **110** and **112** in a single cycle. Sharing of the data cache **106** among independent processor elements **110** and **112** advantageously simplifies data handling, avoiding a need for a cache coordination protocol and the overhead incurred in controlling the protocol.

In addition to the instruction cache **210**, the instruction supply blocks in an execution path include the instruction aligner **212**, and the instruction buffer **214** that precisely format and align a full instruction group of four instructions to prepare to access the register file **216**. An individual execution path has a single register file **216** that is physically split into multiple register file segments, each of which is associated with a particular

functional unit of the multiple functional units. At any point in time, the register file segments as allocated to each functional unit each contain the same content. A multi-ported register file is typically metal limited to the area consumed by the circuit proportional with the square of the number of ports. The processor **100** has a register file structure divided into a plurality of separate and independent register files to form a layout structure with an improved layout efficiency. The read ports of the total register file structure **216** are allocated among the separate and individual register files. Each of the separate and individual register files has write ports that correspond to the total number of write ports in the total register file structure. Writes are fully broadcast so that all of the separate and individual register files are coherent.

The media functional units **220** are multiple single-instruction-multiple-datapath (MSIMD) media functional units. Each of the media functional units **220** is capable of processing parallel 16-bit components. Various parallel 16-bit operations supply the single-instruction-multiple-datapath capability for the processor **100** including add, multiply-add, shift, compare, and the like. The media functional units **220** operate in combination as tightly coupled digital signal processors (DSPs). Each media functional unit **220** has an separate and individual sub-instruction stream, but all three media functional units **220** execute synchronously so that the subinstructions progress lock-step through pipeline stages.

The general functional unit **222** is a RISC processor capable of executing arithmetic logic unit (ALU) operations, loads and stores, branches, and various specialized and esoteric functions such as parallel power operations, reciprocal square root operations, and many others. The general functional unit **222** supports less common parallel operations such as the parallel reciprocal square root instruction.

The illustrative instruction cache **210** is two-way set-associative, has a 16 Kbyte capacity, and includes hardware support to maintain coherence, allowing dynamic optimizations through self-modifying code. Software is used to indicate that the instruction storage is being modified when modifications occur. The 16K capacity is suitable for performing graphic loops, other multimedia tasks or processes, and general-purpose Java™ code. Coherency is maintained by hardware that supports write-through, non-allocating caching. Self-modifying code is supported through explicit use of "store-to-instruction-space" instruction *store2i*. Software uses the *store2i* instruction to maintain coherency with the instruction cache **210** so that the instruction caches **210** do not have to be snooped on every single store operation issued by the media processing unit **110**.

The pipeline control unit **226** is connected between the instruction buffer **214** and the functional units and schedules the transfer of instructions to the functional units. The pipeline control unit **226** also receives status signals from the functional units and the load/store unit **218** and uses the status signals to perform several control functions. The pipeline control unit **226** maintains a scoreboard, generates stalls and bypass controls. The pipeline control unit **226** also generates traps and maintains special registers.

Each media processing unit **110** and **112** includes a split register file **216**, a single logical register file including 128 thirty-two bit registers. The split register file **216** is split into a plurality of register file segments **224** to form a multi-ported structure that is replicated to reduce the integrated circuit die area and to reduce access time. A separate register file segment **224** is allocated to each of the media functional units **220** and the general functional unit **222**. In the illustrative embodiment, each register file segment **224** has 128 32-bit registers. The first 96 registers (0-95) in the register file segment **224** are global registers. All functional units can write to the 96 global registers. The global registers are coherent across all functional units (MFU and GFU) so that any write operation to a global register by any functional unit is broadcast to all register file segments **224**. Registers 96-127 in the register file segments **224** are local registers. Local registers allocated to a functional unit are not accessible or "visible" to other functional units.

The media processing units **110** and **112** are highly structured computation blocks that execute software-scheduled data computation operations with fixed, deterministic and relatively short instruction latencies, operational characteristics yielding simplification in both function and cycle time. The operational characteristics support multiple instruction issue through a pragmatic very large instruction word (VLIW) approach that avoids hardware interlocks to account for software that does not schedule operations properly. Such hardware interlocks are typically complex, error-prone, and create multiple critical paths. A VLIW instruction word always includes one instruction that executes in the general functional unit (GFU) **222** and from zero to three instructions that execute in the media functional units (MFU) **220**. A MFU instruction field within the VLIW instruction word includes an operation code (opcode) field, three source register (or immediate) fields, and one destination register field.

Instructions are executed in-order in the processor **100** but loads can finish out-of-order with respect to other instructions and with respect to other loads, allowing loads to be moved up in the instruction stream so that data can be streamed from main memory. The execution model eliminates the usage and overhead resources of an instruction window, reservation stations, a re-order buffer, or other blocks for handling instruction ordering. Elimination of the instruction ordering structures and overhead resources is highly advantageous since the eliminated blocks typically consume a large portion of an integrated circuit die. For example, the eliminated blocks consume about 30% of the die area of a Pentium II processor.

To avoid software scheduling errors, the media processing units **110** and **112** are high-performance but simplified with respect to both compilation and execution. The media processing units **110** and **112** are most generally classified as a simple 2-scalar execution engine with full bypassing and hardware interlocks on load operations. The instructions include loads, stores, arithmetic and logic (ALU) instructions, and branch instructions so that scheduling for the processor **100** is essentially equivalent to scheduling for a simple 2-scalar execution engine for each of the two media processing units **110** and **112**.

The processor **100** supports full bypasses between the first two execution units within the media processing unit **110** and **112** and has a scoreboard in the general functional unit **222** for load operations so that

the compiler does not need to handle nondeterministic latencies due to cache misses. The processor **100** scoreboards long latency operations that are executed in the general functional unit **222**, for example a reciprocal square-root operation, to simplify scheduling across execution units. The scoreboard (not shown) operates by tracking a record of an instruction packet or group from the time the instruction enters a functional unit until the instruction is finished and the result becomes available. A VLIW instruction packet contains one GFU instruction and from zero to three MFU instructions. The source and destination registers of all instructions in an incoming VLIW instruction packet are checked against the scoreboard. Any true dependencies or output dependencies stall the entire packet until the result is ready. Use of a scoreboarded result as an operand causes instruction issue to stall for a sufficient number of cycles to allow the result to become available. If the referencing instruction that provokes the stall executes on the general functional unit **222** or the first media functional unit **220**, then the stall only endures until the result is available for intra-unit bypass. For the case of a *load* instruction that hits in the data cache **106**, the stall may last only one cycle. If the referencing instruction is on the second or third media functional units **220**, then the stall endures until the result reaches the writeback stage in the pipeline where the result is bypassed in transmission to the split register file **216**.

The scoreboard automatically manages load delays that occur during a load hit. In an illustrative embodiment, all loads enter the scoreboard to simplify software scheduling and eliminate NOPs in the instruction stream.

The scoreboard is used to manage most interlock conditions between the general functional unit **222** and the media functional units **220**. All loads and non-pipelined long-latency operations of the general functional unit **222** are scoreboarded. The long-latency operations include division *idiv*, *fdiv* instructions, reciprocal square root *frecsqrt*, *precsqrt* instructions, and power *ppower* instructions. None of the results of the media functional units **220** is scoreboarded. Non-scoreboarded results are available to subsequent operations on the functional unit that produces the results following the latency of the instruction.

The illustrative processor **100** has a rendering rate of over fifty million triangles per second without accounting for operating system overhead. Therefore, data feeding specifications of the processor **100** are far beyond the capabilities of cost-effective memory systems. Sufficient data bandwidth is achieved by rendering of compressed geometry using the geometry decompressor **104**, an on-chip real-time geometry decompression engine. Data geometry is stored in main memory in a compressed format. At render time, the data geometry is fetched and decompressed in real-time on the integrated circuit of the processor **100**. The geometry decompressor **104** advantageously saves memory space and memory transfer bandwidth. The compressed geometry uses an optimized generalized mesh structure that explicitly calls out most shared vertices between triangles, allowing the processor **100** to transform and light most vertices only once. In a typical compressed mesh, the triangle throughput of the transform-and-light stage is increased by a factor of four or more over the throughput for isolated triangles. For example, during processing of triangles, multiple vertices are operated

upon in parallel so that the utilization rate of resources is high, achieving effective spatial software pipelining. Thus operations are overlapped in time by operating on several vertices simultaneously, rather than overlapping several loop iterations in time. For other types of applications with high instruction level parallelism, high trip count loops are software-pipelined so that most media functional units **220** are fully utilized.

Referring to **FIGURE 3**, a schematic block diagram illustrates an embodiment of the split register file **216** that is suitable for usage in the processor **100**. The split register file **216** supplies all operands of processor instructions that execute in the media functional units **220** and the general functional units **222** and receives results of the instruction execution from the execution units. The split register file **216** operates as an interface to the geometry decompressor **104**. The split register file **216** is the source and destination of store and load operations, respectively.

In the illustrative processor **100**, the split register file **216** in each of the media processing units **110** and **112** has 128 registers. Graphics processing places a heavy burden on register usage. Therefore, a large number of registers is supplied by the split register file **216** so that performance is not limited by loads and stores or handling of intermediate results including graphics "fills" and "spills". The illustrative split register file **216** includes twelve read ports and five write ports, supplying total data read and write capacity between the central registers of the split register file **216** and all media functional units **220** and the general functional unit **222**. The five write ports include one 64-bit write port that is dedicated to load operations. The remaining four write ports are 32 bits wide and are used to write operations of the general functional unit **222** and the media functional units **220**.

A large total read and write capacity promotes flexibility and facility in programming both of hand-coded routines and compiler-generated code.

Large, multiple-ported register files are typically metal-limited so that the register area is proportional with the square of the number of ports. A sixteen port file is roughly proportional in size and speed to a value of 256. The illustrative split register file **216** is divided into four register file segments **310, 312, 314**, and **316**, each having three read ports and four write ports so that each register file segment has a size and speed proportional to 49 for a total area for the four segments that is proportional to 196. The total area is therefore potentially smaller and faster than a single central register file. Write operations are fully broadcast so that all files are maintained coherent. Logically, the split register file **216** is no different from a single central register file. However, from the perspective of layout efficiency, the split register file **216** is highly advantageous, allowing for reduced size and improved performance.

The new media data that is operated upon by the processor **100** is typically heavily compressed. Data transfers are communicated in a compressed format from main memory and input/output devices to pins of the

processor **100**, subsequently decompressed on the integrated circuit holding the processor **100**, and passed to the split register file **216**.

Splitting the register file into multiple segments in the split register file **216** in combination with the character of data accesses in which multiple bytes are transferred to the plurality of execution units concurrently, results in a high utilization rate of the data supplied to the integrated circuit chip and effectively leads to a much higher data bandwidth than is supported on general-purpose processors. The highest data bandwidth requirement is therefore not between the input/output pins and the central processing units, but is rather between the decompressed data source and the remainder of the processor. For graphics processing, the highest data bandwidth requirement is between the geometry decompressor **104** and the split register file **216**. For video decompression, the highest data bandwidth requirement is internal to the split register file **216**. Data transfers between the geometry decompressor **104** and the split register file **216** and data transfers between various registers of the split register file **216** can be wide and run at processor speed, advantageously delivering a large bandwidth.

The register file **216** is a focal point for attaining the very large bandwidth of the processor **100**. The processor **100** transfers data using a plurality of data transfer techniques. In one example of a data transfer technique, cacheable data is loaded into the split register file **216** through normal load operations at a low rate of up to eight bytes per cycle. In another example, streaming data is transferred to the split register file **216** through group load operations, which transfer thirty-two bytes from memory directly into eight consecutive 32-bit registers. The processor **100** utilizes the streaming data operation to receive compressed video data for decompression.

Compressed graphics data is received via a direct memory access (DMA) unit in the geometry decompressor **104**. The compressed graphics data is decompressed by the geometry decompressor **104** and loaded at a high bandwidth rate into the split register file **216** via group load operations that are mapped to the geometry decompressor **104**.

Load operations are non-blocking and scoreboarded so that early scheduling can hide a long latency inherent to loads.

General purpose applications often fail to exploit the large register file **216**. Statistical analysis shows that compilers do not effectively use the large number of registers in the split register file **216**. However, aggressive in-lining techniques that have traditionally been restricted due to the limited number of registers in conventional systems may be advantageously used in the processor **100** to exploit the large number of registers in the split register file **216**. In a software system that exploits the large number of registers in the processor **100**, the complete set of registers is saved upon the event of a thread (context) switch. When only a few registers of the entire set of registers is used, saving all registers in the full thread switch is wasteful. Waste is

avoided in the processor **100** by supporting individual marking of registers. Octants of the thirty-two registers can be marked as "dirty" if used, and are consequently saved conditionally.

In various embodiments, dedicating fields for globals, trap registers, and the like leverages the split register file 216.

Referring to **FIGURE 4**, a schematic block diagram shows a logical view of the register file **216** and functional units in the processor **100**. The physical implementation of the core processor **100** is simplified by replicating a single functional unit to form the three media functional units **220**. The media functional units **220** include circuits that execute various arithmetic and logical operations including general-purpose code, graphics code, and video-image-speech (VIS) processing. VIS processing includes video processing, image processing, digital signal processing (DSP) loops, speech processing, and voice recognition algorithms, for example.

Referring to **FIGURE 5**, a simplified pictorial schematic diagram depicts an example of instruction execution among a plurality of media functional units **220**. Results generated by various internal function blocks within a first individual media functional unit are immediately accessible internally to the first media functional unit **510** but are only accessible globally by other media functional units **512** and **514** and by the general functional unit five cycles after the instruction enters the first media functional unit **510**, regardless of the actual latency of the instruction. Therefore, instructions executing within a functional unit can be scheduled by software to execute immediately, taking into consideration the actual latency of the instruction. In contrast, software that schedules instructions executing in different functional units is expected to account for the five cycle latency. In the diagram, the shaded areas represent the stage at which the pipeline completes execution of an instruction and generates final result values. A result is not available internal to a functional unit a final shaded stage completes. In the example, media processing unit instructions have three different latencies - four cycles for instructions such as fmuladd and fadd, two cycles for instructions such as pmuladd, and one cycle for instructions like padd and xor.

Although internal bypass logic within a media functional unit **220** forwards results to execution units within the same media functional unit **220**, the internal bypass logic does not detect incorrect attempts to reference a result before the result is available.

Software that schedules instructions for which a dependency occurs between a particular media functional unit, for example **512**, and other media functional units **510** and **514**, or between the particular media functional unit **512** and the general functional unit **222**, is to account for the five cycle latency between entry of an instruction to the media functional unit **512** and the five cycle pipeline duration.

Referring to **FIGURE 6**, a schematic block diagram depicts an embodiment of the multiport register file **216**. A plurality of read address buses **RA1** through **RAN** carry read addresses that are applied to decoder

ports **616-1** through **616-N**, respectively. Decoder circuits are well known to those of ordinary skill in the art, and any of several implementations could be used as the decoder ports **616-1** through **616-N**. When an address is presented to any of decoder ports **616-1** through **616-N**, the address is decoded and a read address signal is transmitted by a decoder port **616** to a register in a memory cell array **618**. Data from the memory cell array **618** is output using output data drivers **622**. Data is transferred to and from the memory cell array **618** under control of control signals carried on some of the lines of the buses of the plurality of read address buses **RA1** through **RAN**.

Referring to **FIGURE 7A** and **7B**, a schematic block diagram and a pictorial diagram, respectively, illustrate the register file **216** and a memory array insert **710**. The register file **216** is connected to a four functional units **720**, **722**, **724**, and **726** that supply information for performing operations such as arithmetic, logical, graphics, data handling operations and the like. The illustrative register file **216** has twelve read ports **730** and four write ports **732**. The twelve read ports **730** are illustratively allocated with three ports connected to each of the four functional units. The four write ports **732** are connected to receive data from all of the four functional units.

The register file **216** includes a decoder, as is shown in **FIGURE 6**, for each of the sixteen read and write ports. The register file **216** includes a memory array **740** that is partially shown in the insert **710** illustrated in **FIGURE 7B** and includes a plurality of word lines **744** and bit lines **746**. The word lines **744** and bit lines **746** are simply a set of wires that connect transistors (not shown) within the memory array **740**. The word lines **744** select registers so that a particular word line selects a register of the register file **216**. The bit lines **746** are a second set of wires that connect the transistors in the memory array **740**. Typically, the word lines **744** and bit lines **746** are laid out at right angles. In the illustrative embodiment, the word lines **744** and the bit lines **746** are constructed of metal laid out in different planes such as a metal 2 layer for the word lines **744** and a metal 3 layer for the bit lines **746**. In other embodiments, bit lines and word lines may be constructed of other materials, such as polysilicon, or can reside at different levels than are described in the illustrative embodiment, that are known in the art of semiconductor manufacture. In the illustrative example, a distance of about 1 μm separates the word lines **744** and a distance of approximately 1 μm separates the bit lines **746**. Other circuit dimensions may be constructed for various processes. The illustrative example shows one bit line per port, other embodiments may use multiple bit lines per port.

When a particular functional unit reads a particular register in the register file **216**, the functional unit sends an address signal via the read ports **730** that activates the appropriate word lines to access the register. In a register file having a conventional structure and twelve read ports, each cell, each storing a single bit of information, is connected to twelve word lines to select an address and twelve bit lines to carry data read from the address.

The four write ports 732 address registers in the register file using four word lines 744 and four bit lines 746 connected to each cell. The four word lines 744 address a cell and the four bit lines 746 carry data to the cell.

Thus, if the illustrative register file 216 were laid out in a conventional manner with twelve read ports 730 and four write ports 732 for a total of sixteen ports and the ports were 1μm apart, one memory cell would have an integrated circuit area of $256\mu m^2$ (16x16). The area is proportional to the square of the number of ports.

The register file 216 is alternatively implemented to perform single-ended reads and/or single-ended writes utilizing a single bit line per port per cell, or implemented to perform differential reads and/or differential writes using two bit lines per port per cell.

However, in this embodiment the register file 216 is not laid out in the conventional manner and instead is split into a plurality of separate and individual register file segments 224. Referring to **FIGURE 8**, a schematic block diagram shows an arrangement of the register file 216 into the four register file segments 224. The register file 216 remains operational as a single logical register file in the sense that the four of the register file segments 224 contain the same number of registers and the same register values as a conventional register file of the same capacity that is not split. The separated register file segments 224 differ from a register file that is not split through elimination of lines that would otherwise connect ports to the memory cells. Accordingly, each register file segment 224 has connections to only three of the twelve read ports 730, lines connecting a register file segment to the other nine read ports are eliminated. All writes are broadcast so that each of the four register file segments 224 has connections to all four write ports 732. Thus each of the four register file segments 224 has three read ports and four write ports for a total of seven ports. The individual cells are connected to seven word lines and seven bit lines so that a memory array with a spacing of 1μm between lines has an area of approximately $49\mu m^2$. In the illustrative embodiment, the four register file segments 224 have an area proportion to seven squared. The total area of the four register file segments 224 is therefore proportional to 49 times 4, a total of 196.

The split register file thus advantageously reduces the area of the memory array by a ratio of approximately 256/196 (1.3X or 30%). The reduction in area further advantageously corresponds to an improvement in speed performance due to a reduction in the length of the word lines 744 and the bit lines 746 connecting the array cells that reduces the time for a signal to pass on the lines. The improvement in speed performance is highly advantageous due to strict time budgets that are imposed by the specification of high-performance processors and also to attain a large capacity register file that is operational at high speed. For example, the operation of reading the register file 216 typically takes place in a single clock cycle. For a processor that executes at 500 MHz, a cycle time of two nanoseconds is imposed for accessing the register file 216. Conventional register files typically only have up to about 32 registers in comparison to the 128 registers

in the illustrative register file **216** of the processor **100**. A register file **216** substantially larger than the register file in conventional processors is highly advantageous in high-performance operations such as video and graphic processing. The reduced size of the register file **216** is highly useful for complying with time budgets in a large capacity register file.

Referring to **FIGURE 9**, a simplified schematic timing diagram illustrates timing of the processor pipeline **900**. The pipeline **900** includes nine stages including three initiating stages, a plurality of execution phases, and two terminating stages. The three initiating stages are optimized to include only those operations necessary for decoding instructions so that jump and call instructions, which are pervasive in the Java^{TM} language, execute quickly. Optimization of the initiating stages advantageously facilitates branch prediction since branches, jumps, and calls execute quickly and do not introduce many bubbles.

The first of the initiating stages is a fetch stage **910** during which the processor **100** fetches instructions from the 16Kbyte two-way set-associative instruction cache **210**. The fetched instructions are aligned in the instruction aligner **212** and forwarded to the instruction buffer **214** in an align stage **912**, a second stage of the initiating stages. The aligning operation properly positions the instructions for storage in a particular segment of the four register file segments **310, 312, 314,** and **316** and for execution in an associated functional unit of the three media functional units **220** and one general functional unit **222**. In a third stage, a decoding stage **914** of the initiating stages, the fetched and aligned VLIW instruction packet is decoded and the scoreboard (not shown) is read and updated in parallel. The four register file segments **310, 312, 314, and 316** each holds either floating-point data or integer data. The register files are read in the decoding (D) stage.

Following the decoding stage **914**, the execution stages are performed. The two terminating stages include a trap-handling stage **960** and a write-back stage **962** during which result data is written-back to the split register file **216**.

While the invention has been described with reference to various embodiments, it will be understood that these embodiments are illustrative and that the scope of the invention is not limited to them. Many variations, modifications, additions and improvements of the embodiments described are possible. For example, those skilled in the art will readily implement the steps necessary to provide the structures and methods disclosed herein, and will understand that the process parameters, materials, and dimensions are given by way of example only and can be varied to achieve the desired structure as well as modifications which are within the scope of the invention. Variations and modifications of the embodiments disclosed herein may be made based on the description set forth herein, without departing from the scope and spirit of the invention as set forth in the following claims.

For example, while the illustrative embodiment specifically discusses advantages gained in using the Java^{TM} programming language with the described system, any suitable programming language is also supported. Other programming languages that support multiple-threading are generally more advantageously used in the

described system. Also, while the illustrative embodiment specifically discusses advantages attained in using Java Virtual Machines™ with the described system, any suitable processing engine is also supported. Other processing engines that support multiple-threading are generally more advantageously used in the described system.

Furthermore, although the illustrative register file has one bit line per port, in other embodiments more bit lines may be allocated for a port. The described word lines and bit lines are formed of a metal. In other examples, other conductive materials such as doped polysilicon may be employed for interconnects. The described register file uses single-ended reads and writes so that a single bit line is employed per bit and per port. In other processors, differential reads and writes with dual-ended sense amplifiers may be used so that two bit lines are allocated per bit and per port, resulting in a bigger pitch. Dual-ended sense amplifiers improve memory fidelity but greatly increase the size of a memory array, imposing a heavy burden on speed performance. Thus the advantages attained by the described register file structure are magnified for a memory using differential reads and writes. The spacing between bit lines and word lines is described to be approximately 1μm. In some processors, the spacing may be greater than 1μm. In other processors the spacing between lines is less than 1μm.

WE CLAIM

1. A processor comprising:
a plurality of independent parallel execution paths that execute in parallel across a plurality of threads, the execution paths including a multiple instruction parallel pathway within a thread; and
the independent parallel execution paths including functional units that execute an instruction set including special data handling instructions supporting a multiple-thread execution environment.

2. A processor according to Claim 1 wherein:
the plurality of independent parallel instruction paths execute as a plurality of processors in multiple-threaded applications using a Java™ programming language running under a multiple-threaded operating system on a multiple-threaded Java Virtual Machine™.

3. A processor according to Claim 1 wherein:
the processor includes two independent processor elements forming a respective two independent parallel execution paths.

4. A processor according to Claim 1 wherein:
the plurality of independent parallel instruction paths execute as a plurality of processors in multiple-threaded applications using a Java™ programming language that generates a plurality of threads that respectively execute in the plurality of independent parallel instruction paths with a minimum of threading overhead.

5. A processor according to Claim 1 wherein:
the plurality of independent parallel instruction paths execute as a plurality of processors in multiple-threaded applications using a Java™ programming language supporting special instructions for accessing arrays and instructions supporting garbage collection.

6. A processor according to Claim 1 wherein:
the independent processor elements are Very Long Instruction Word (VLIW) processors forming a respective plurality of independent parallel execution paths.

7. A processor according to Claim 1 wherein:
the independent processor elements are integrated into a single integrated-circuit chip.

8. A processor comprising:

a plurality of independent processor elements in a single integrated circuit chip capable of executing a respective plurality of threads concurrently during a multiple-threaded operation.

9. A processor according to Claim 8 wherein:

the independent processor elements are Very Long Instruction Word (VLIW) processors forming a respective plurality of independent parallel execution paths.

10. A processor according to Claim 8 wherein:

the processor is a general-purpose processor.

11. A processor according to Claim 8 wherein:

the processor includes two independent processor elements in a single integrated circuit chip.

12. A processor according to Claim 8 wherein:

the independent processor elements include a plurality of functional units that execute a respective plurality of instructions concurrently and in parallel.

13. A processor according to Claim 8 wherein:

a plurality of independent processor elements are Very Long Instruction Word (VLIW) processor elements that include a plurality of functional units operating concurrently in parallel, the functional units including media functional units operating as digital signal processors, and a general functional unit, and

the media functional units capable of executing a instruction that executes both a multiply operation and an addition operation in a single cycle, the multiply operation and add operations being either floating point or fixed point.

14. A processor comprising:

a plurality of independent processor elements in a single concurrently executable parallel processor, the independent processor elements including:

an instruction supply logic;

an instruction preparation logic coupled to the instruction supply logic;

a plurality of functional units coupled to the instruction supply logic and coupled to the instruction preparation logic;

a register file coupled to the plurality of functional units, coupled to the instruction supply logic, and coupled to the instruction preparation logic,

the instruction supply logic, the instruction preparation logic, the plurality of functional units, and the register file for a first independent processor element being independent and separate from the instruction supply logic, the instruction preparation logic, the plurality of functional units, and the register file of a second independent processor element; and

a data cache coupled to and shared among the plurality of independent processor elements.

15. A processor according to Claim 14 wherein:

the plurality of independent processor elements are capable of executing a respective plurality of threads concurrently during a multiple-threaded operation.

16. A processor according to Claim 14 wherein:

the plurality of independent processor elements are integrated into a single integrated-circuit chip.

17. A processor according to Claim 14 wherein:

an instruction supply logic includes an instruction cache for a first independent processor element that is independent and separate from an instruction cache of the instruction supply logic of a second independent processor element

18. A processor according to Claim 14 wherein:

the data cache is multiple-ported, allowing data access in execution paths of the plurality of independent processor elements in a single cycle.

19. A processor according to Claim 14 wherein:

the data cache has a reduced data-handling logic resulting from sharing of the data cache among the plurality of independent processor elements, avoiding necessity of a cache coordination protocol and overhead logic otherwise incurred in controlling the coordination protocol.

20. A processor according to Claim 14 wherein:

an instruction preparation logic includes an aligner and an instruction buffer for a first independent processor element that is independent and separate from an aligner and instruction buffer of the instruction supply logic of a second independent processor element, the aligner that aligns a full instruction group in preparation for accessing the register file.

21. A processor according to Claim 14 wherein:

the register file is physically split into a plurality of register file segments, the individual register file segments being respectively associated and coupled to a functional unit of the plurality of functional units.

22. A processor according to Claim 21 wherein:
the register file has R read ports and W write ports;
the individual register file segments have a reduced number of read ports so that the total number of read ports for the plurality of register file segments is R read ports; and
ones of the individual register file segments have W write ports.

23. A processor according to Claim 21 wherein:
the register file is a sixteen port structure with twelve read ports and five write ports; and
the plurality of register file segments include segments each having three read ports and five write ports.

24. A processor according to Claim 21 wherein:
the register file is a sixteen port structure with twelve read ports and four write ports; and
the plurality of register file segments include segments each having three read ports and four write ports.

25. A processor according to Claim 21 wherein:
write operations are fully broadcast so that all of the separate and individual register files are coherent.

26. A method of operating a processor comprising:
executing in parallel a plurality of execution threads in a plurality of independent parallel execution paths across a plurality of threads;
executing within a thread a plurality of instructions in a multiple-instruction parallel pathway in one of the plurality of independent parallel execution paths; and
executing an instruction set in a plurality of functional units that execute an instruction set including special data handling instructions supporting a multiple-thread environment.

27. A method according to Claim 26 further comprising:
executing the instruction threads in the plurality of independent parallel instruction paths as a plurality of processors in multiple-threaded applications using a Java™ programming language running under a multiple-threaded operating system on a multiple-threaded Java Virtual Machine™.

28. A method according to Claim 26 further comprising:
executing the instruction threads in the plurality of independent parallel instruction paths as a plurality of processors in multiple-threaded applications using a Java™ programming language; and

generating a plurality of threads that respectively execute in the plurality of independent parallel instruction paths with a minimum of threading overhead.

29. A method according to Claim 26 further comprising:
executing the instruction threads in the plurality of independent parallel instruction paths as a plurality of processors in multiple-threaded applications using a Java™ programming language; and
accessing arrays and instructions using special instructions supporting garbage collection.

30. A method according to Claim 26 further comprising:
executing within a thread a plurality of instructions in a multiple-instruction parallel pathway in one of the plurality of independent parallel execution paths using independent processor elements that are Very Long Instruction Word (VLIW) processors.

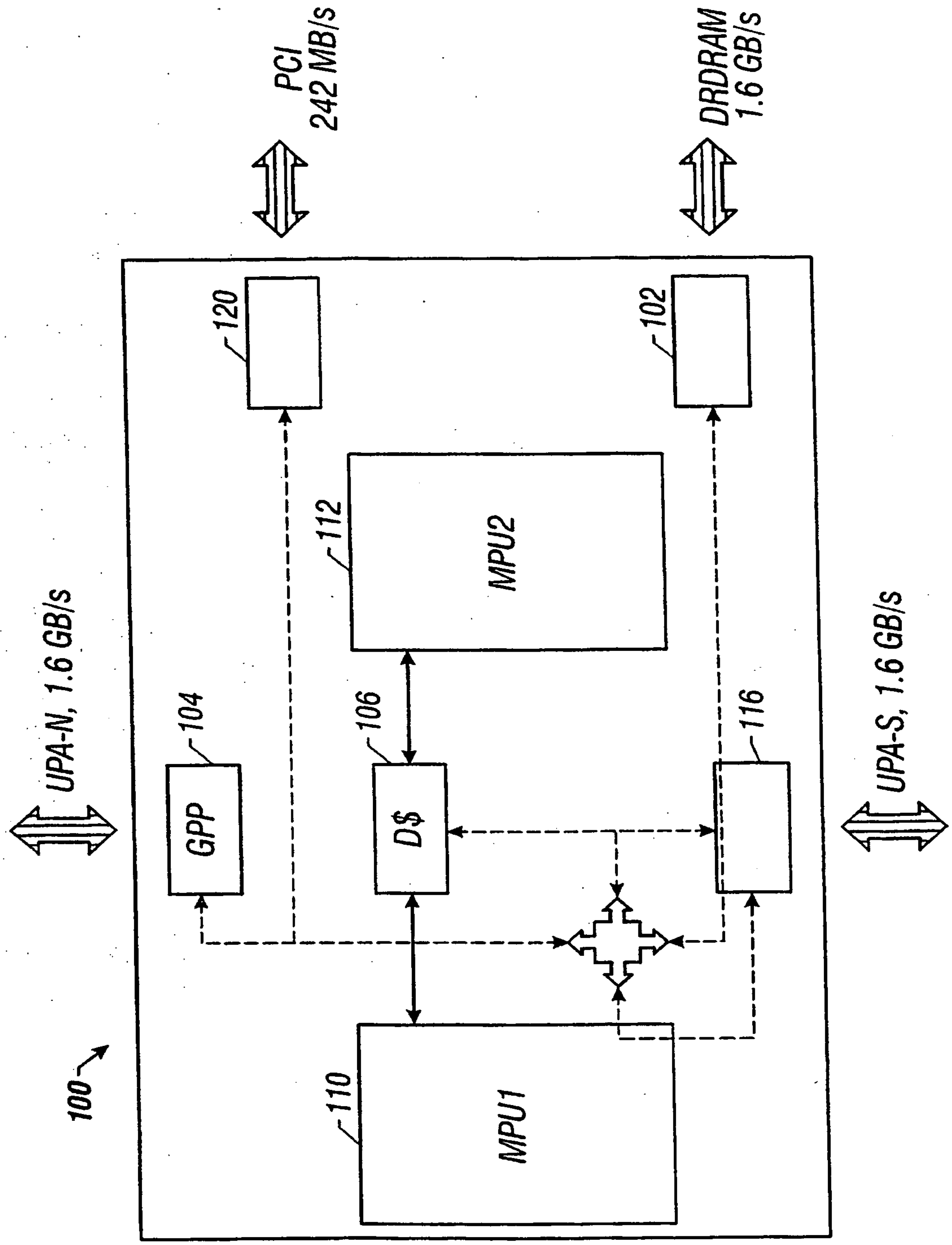

PCI
242 MB/s
DRDRAM
1.6 GB/s
120
102
112
MPU2
UPA-N, 1.6 GB/s
104
GPP
106
D$
116
UPA-S, 1.6 GB/s
100
110
MPU1

FIG. 1

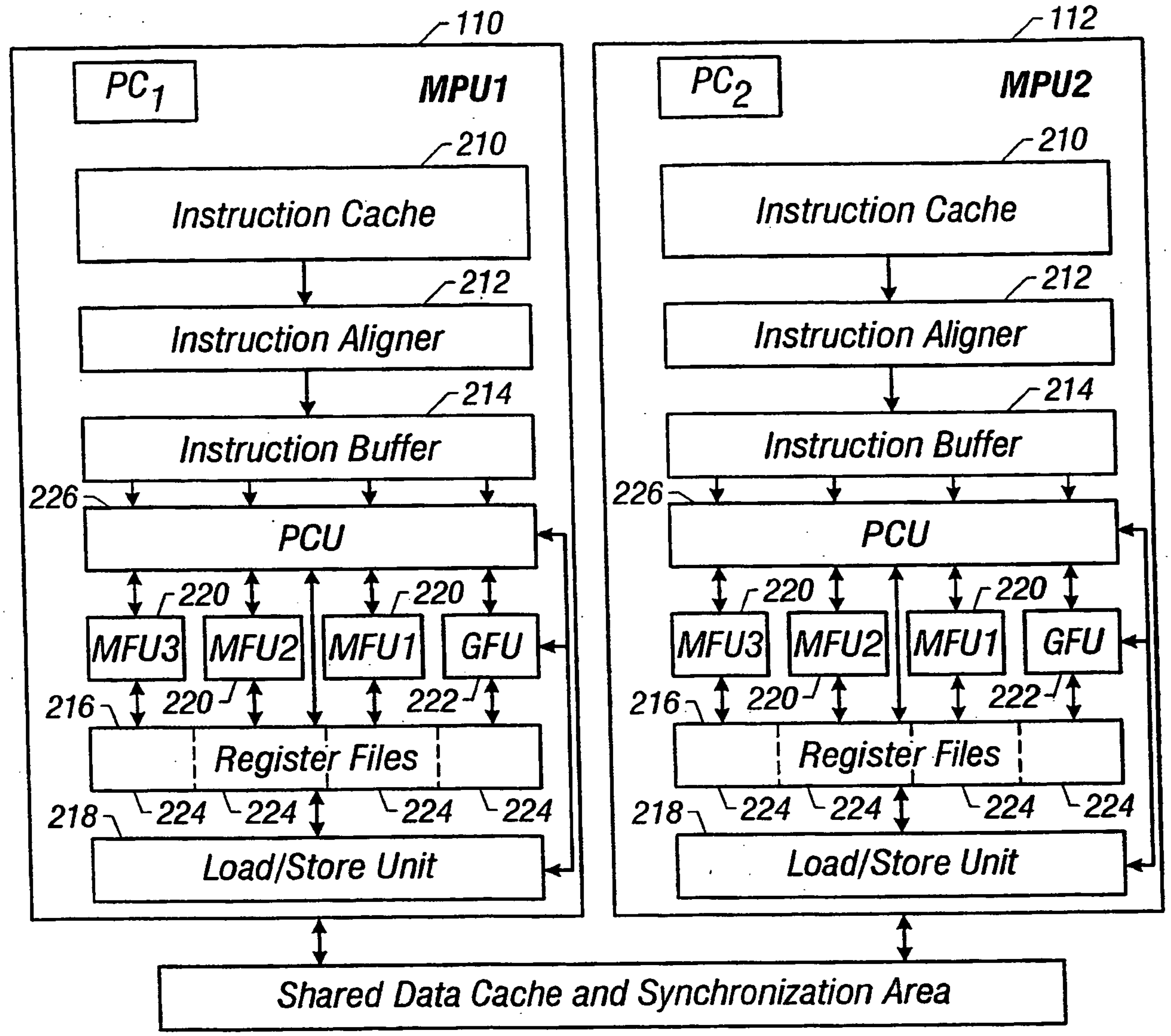

110
PC1
MPU1
210
Instruction Cache
212
Instruction Aligner
214
Instruction Buffer
226
PCU
220
220
MFU3
MFU2
MFU1
GFU
216
220
222
Register Files
218
224
224
224
224
Load/Store Unit
112
PC2
MPU2
210
Instruction Cache
212
Instruction Aligner
214
Instruction Buffer
226
PCU
220
220
MFU3
MFU2
MFU1
GFU
216
220
222
Register Files
218
224
224
224
224
Load/Store Unit
Shared Data Cache and Synchronization Area

FIG. 2

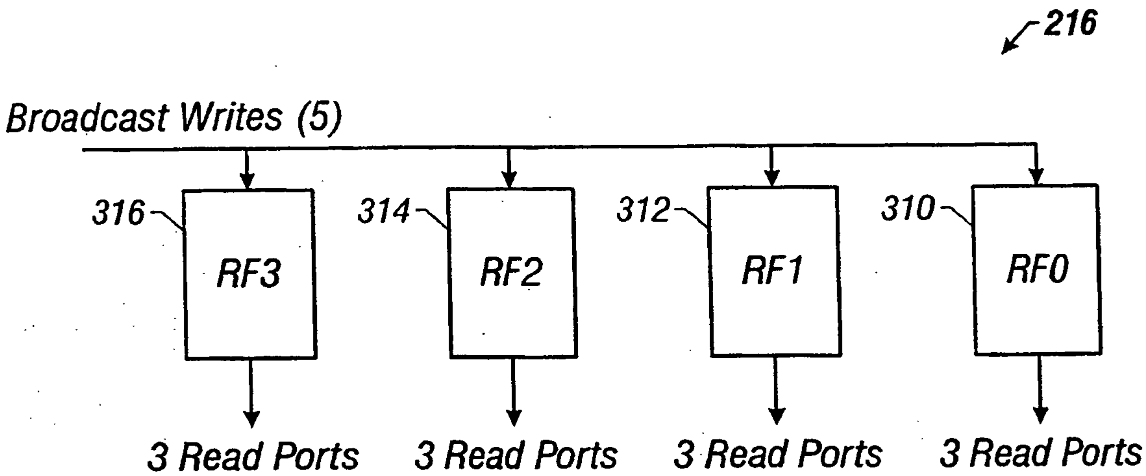

216
Broadcast Writes (5)
316
RF3
314
RF2
312
RF1
310
RF0
3 Read Ports
3 Read Ports
3 Read Ports
3 Read Ports

FIG. 3

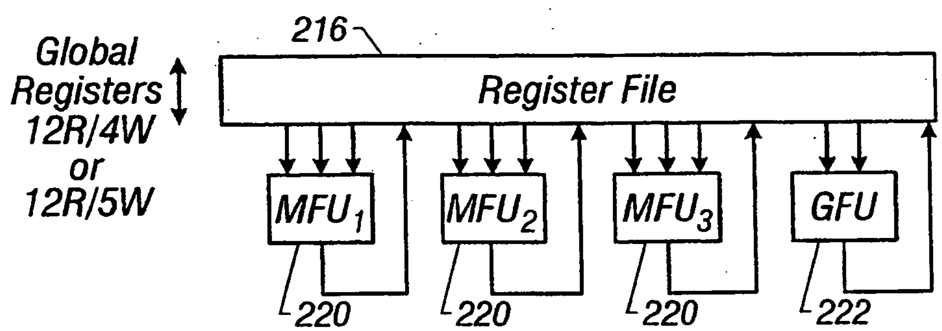

Global Registers 12R/4W or 12R/5W
216
Register File
MFU1
MFU2
MFU3
GFU
220
220
220
222

FIG. 4

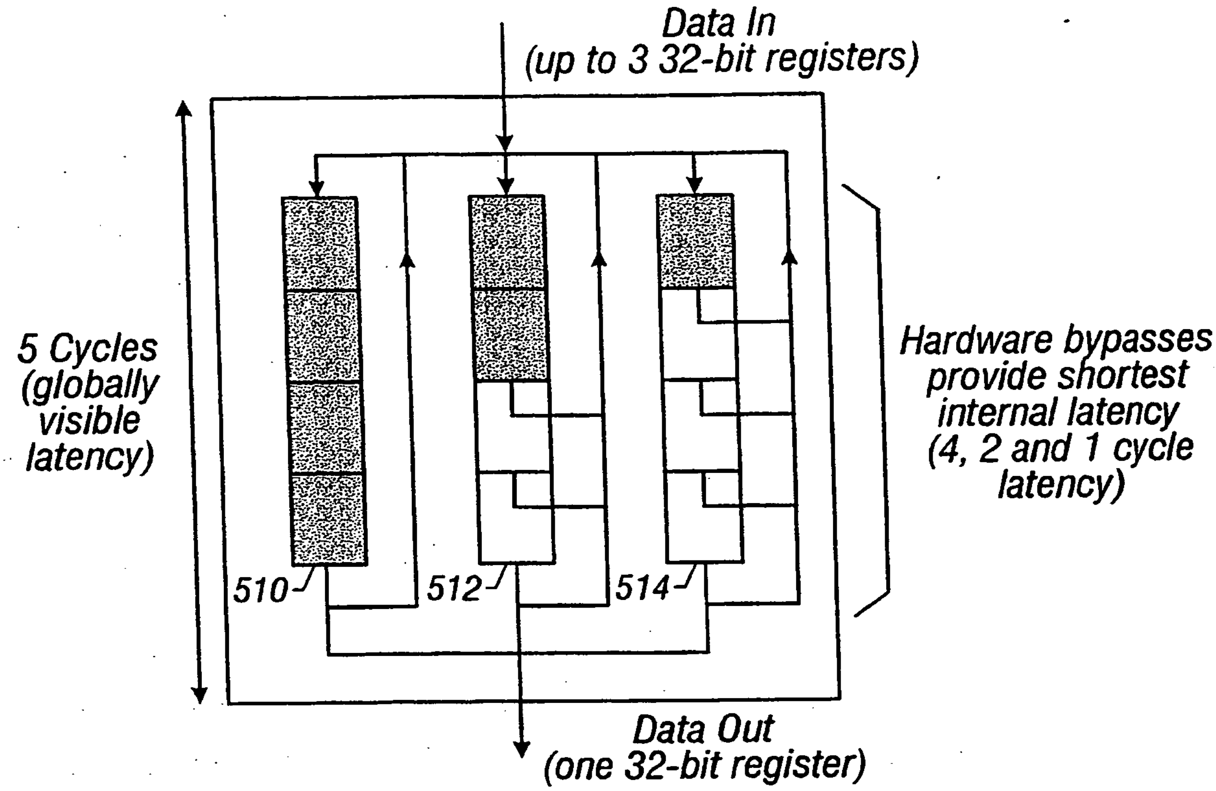

Data In
(up to 3 32-bit registers)
5 Cycles
(globally
visible
latency)
Hardware bypasses
provide shortest
internal latency
(4, 2 and 1 cycle
latency)
510
512
514
Data Out
(one 32-bit register)

FIG. 5

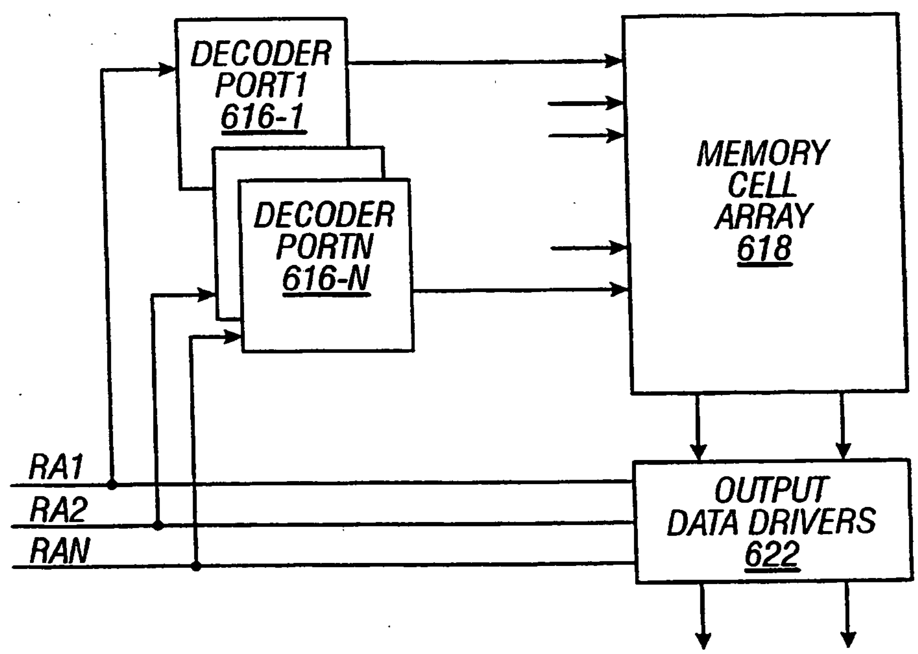

DECODER
PORT1
616-1
DECODER
PORTN
616-N
MEMORY
CELL
ARRAY
618
RA1
RA2
RAN
OUTPUT
DATA DRIVERS
622

FIG. 6

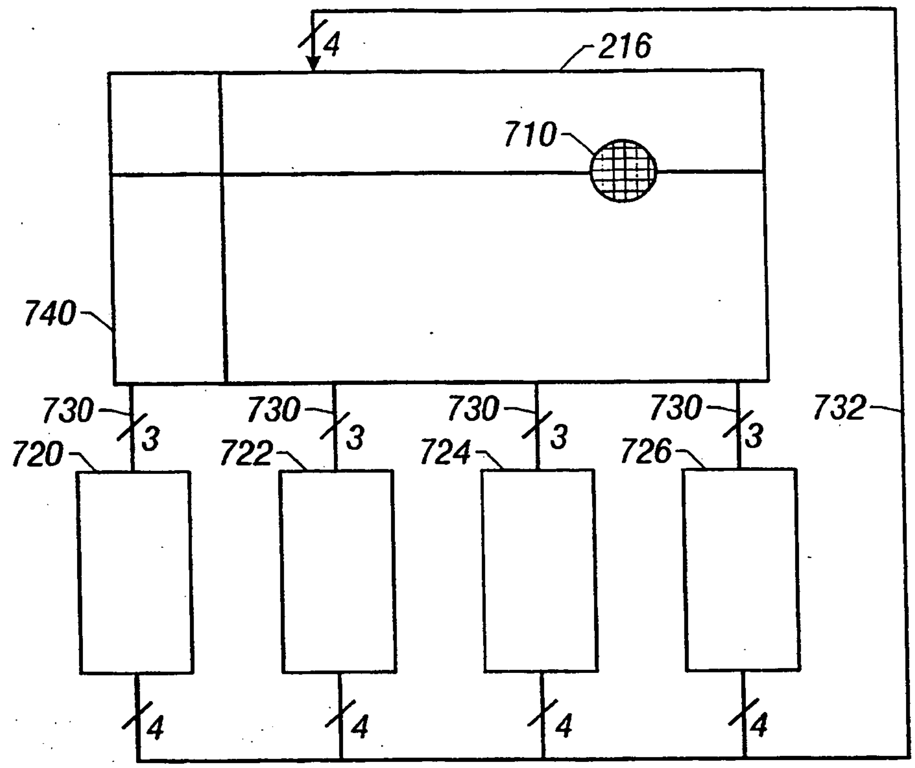

4
216
710
740
730
3
720
722
724
726
732
4

FIG. 7A

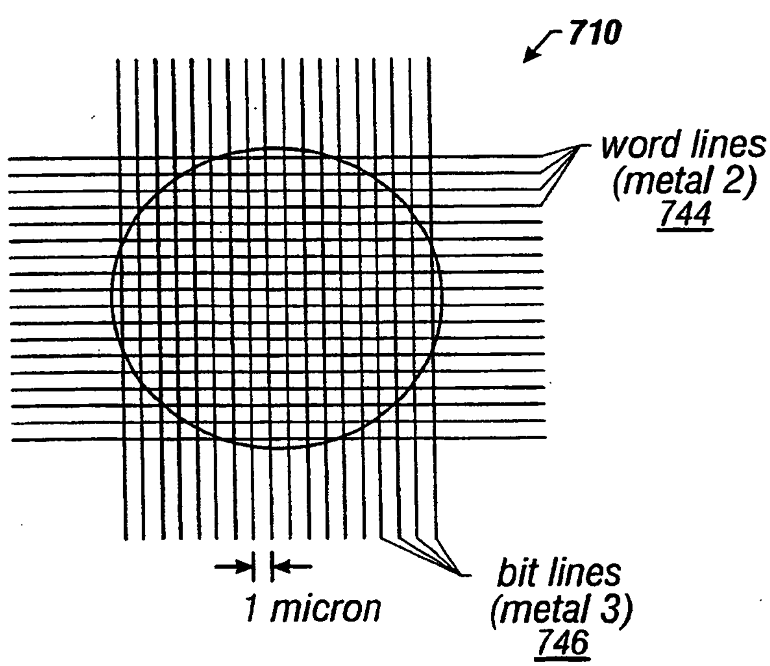

710
word lines
(metal 2)
744
1 micron
bit lines
(metal 3)
746

FIG. 7B

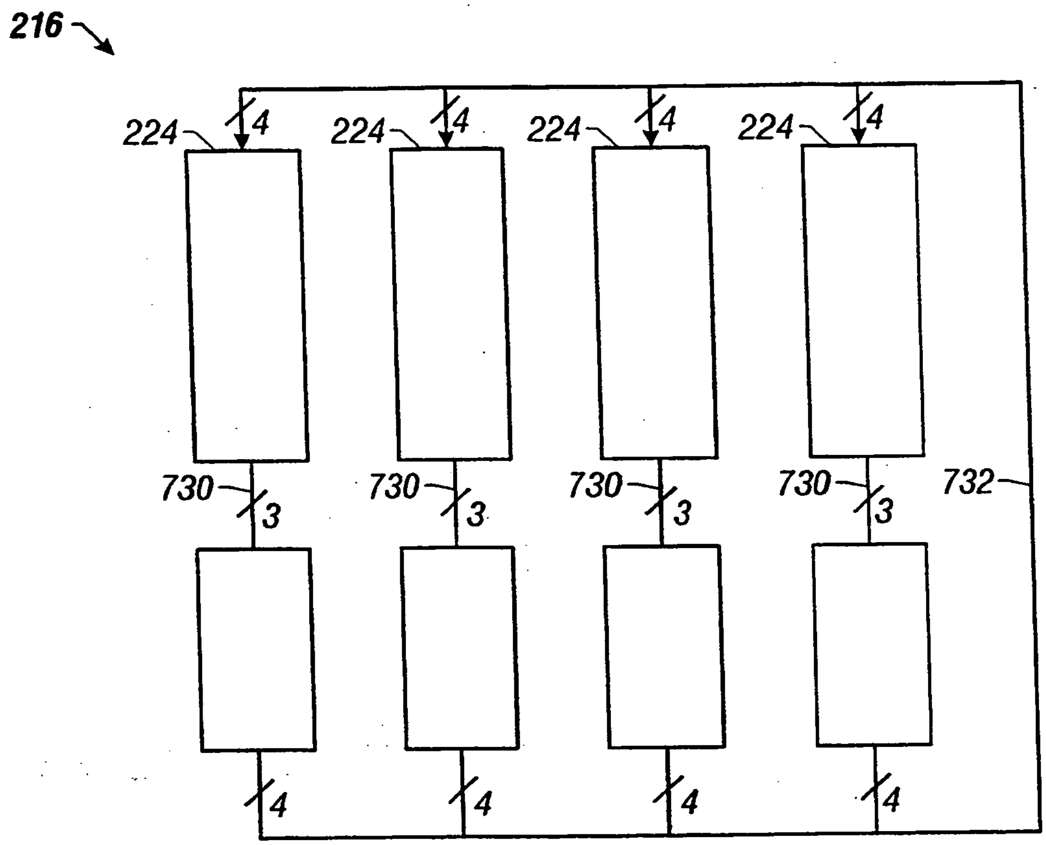


FIG. 8

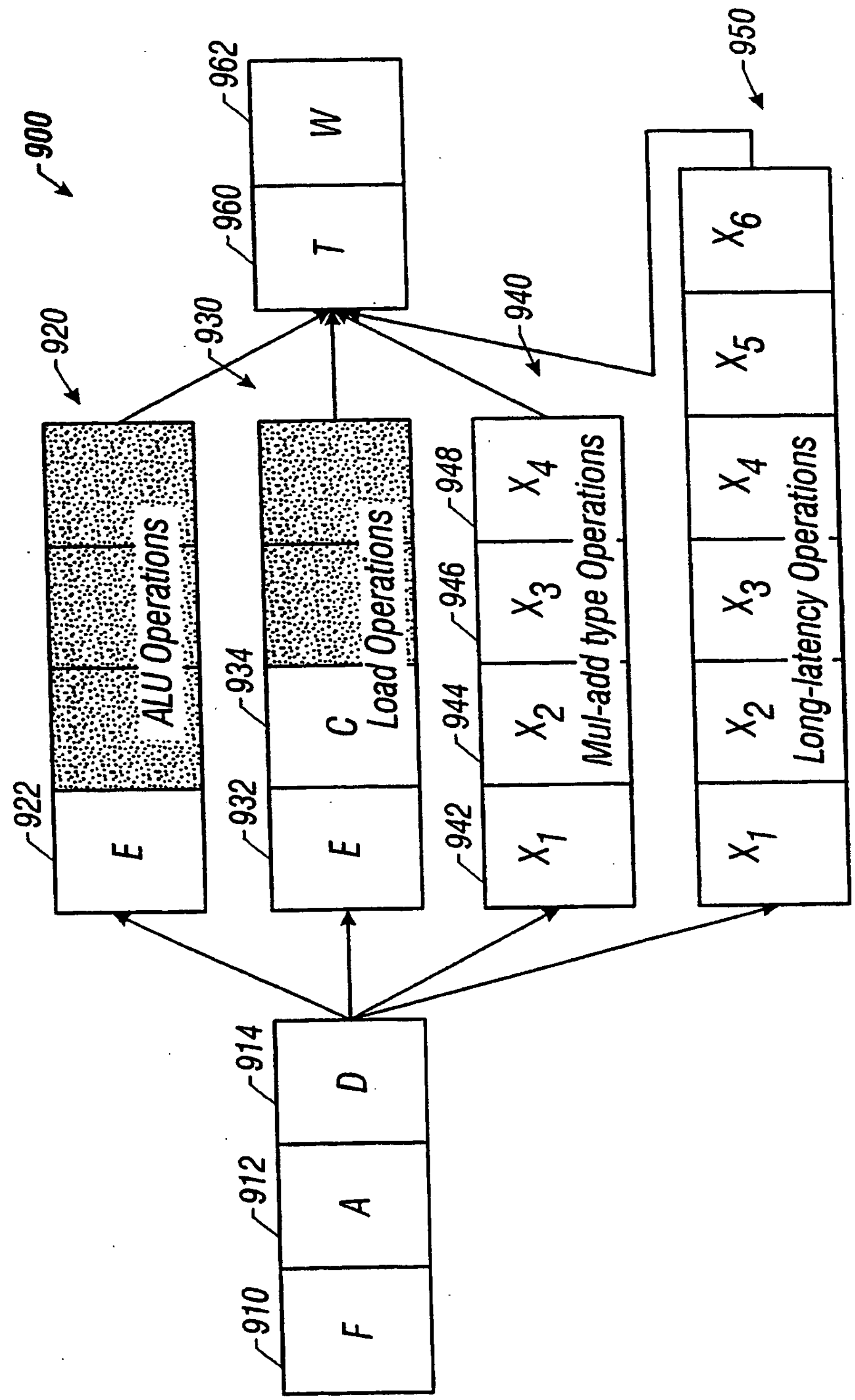

900
910
F
912
A
914
D
920
922
E
ALU Operations
930
932
E
934
C
Load Operations
940
942
X_1
944
X_2
946
X_3
948
X_4
Mul-add type Operations
950
X_1
X_2
X_3
X_4
X_5
X_6
Long-latency Operations
960
T
962
W

FIG. 9

PCT

WORLD INTELLECTUAL PROPERTY ORGANIZATION
International Bureau

INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification 7 : G06F 9/38, 9/46	A3	(11) International Publication Number: WO 00/33185 (43) International Publication Date: 8 June 2000 (08.06.00)

(21) International Application Number: PCT/US99/28821

(22) International Filing Date: 3 December 1999 (03.12.99)

(30) Priority Data:
09/204,480 3 December 1998 (03.12.98) US

(71) Applicant: SUN MICROSYSTEMS, INC. [US/US]; 901 San Antonio Road, M/S PAL01-521, Palo Alto, CA 94303 (US).

(72) Inventors: TREMBLAY, Marc; 140 Hanna Way, Menlo Park, CA 94025 (US). JOY, William, N.; P.O. Box 23, Aspen, CO 81612-0023 (US).

(74) Agents: KOESTNER, Ken, J. et al.; Skjerven, Morrill, MacPherson, Franklin & Friel LLP, Suite 700, 25 Metro Drive, San Jose, CA 95110 (US).

(81) Designated States: JP, KR, European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE).

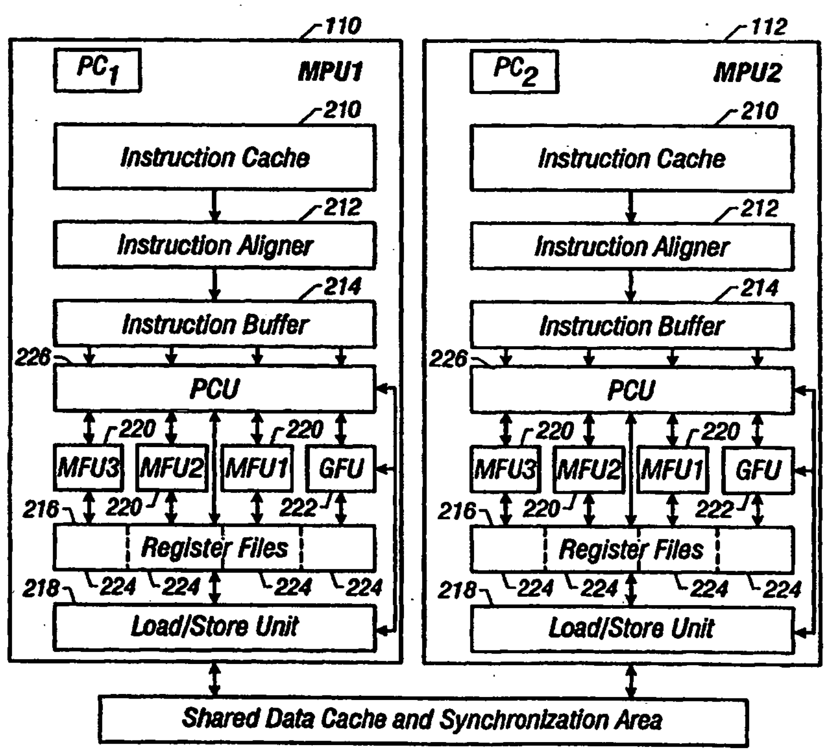
Published
With international search report.

(88) Date of publication of the international search report: 12 October 2000 (12.10.00)

(54) Title: A MULTIPLE-THREAD PROCESSOR FOR THREADED SOFTWARE APPLICATIONS

(57) Abstract

A processor has an improved architecture for multiple-thread operation on the basis of a highly parallel structure including multiple independent parallel execution paths for executing in parallel across threads and a multiple-instruction parallel pathway within a thread. The multiple independent parallel execution paths include functional units that execute an instruction set including special data-handling instructions that are advantageous in a multiple-thread environment.



FOR THE PURPOSES OF INFORMATION ONLY

Codes used to identify States party to the PCT on the front pages of pamphlets publishing international applications under the PCT.

AL	Albania	ES	Spain	LS	Lesotho	SI	Slovenia
AM	Armenia	FI	Finland	LT	Lithuania	SK	Slovakia
AT	Austria	FR	France	LU	Luxembourg	SN	Senegal
AU	Australia	GA	Gabon	LV	Latvia	SZ	Swaziland
AZ	Azerbaijan	GB	United Kingdom	MC	Monaco	TD	Chad
BA	Bosnia and Herzegovina	GE	Georgia	MD	Republic of Moldova	TG	Togo
BB	Barbados	GH	Ghana	MG	Madagascar	TJ	Tajikistan
BE	Belgium	GN	Guinea	MK	The former Yugoslav Republic of Macedonia	TM	Turkmenistan
BF	Burkina Faso	GR	Greece			TR	Turkey
BG	Bulgaria	HU	Hungary	ML	Mali	TT	Trinidad and Tobago
BJ	Benin	IE	Ireland	MN	Mongolia	UA	Ukraine
BR	Brazil	IL	Israel	MR	Mauritania	UG	Uganda
BY	Belarus	IS	Iceland	MW	Malawi	US	United States of America
CA	Canada	IT	Italy	MX	Mexico	UZ	Uzbekistan
CF	Central African Republic	JP	Japan	NE	Niger	VN	Viet Nam
CG	Congo	KE	Kenya	NL	Netherlands	YU	Yugoslavia
CH	Switzerland	KG	Kyrgyzstan	NO	Norway	ZW	Zimbabwe
CI	Côte d'Ivoire	KP	Democratic People's Republic of Korea	NZ	New Zealand		
CM	Cameroon			PL	Poland		
CN	China	KR	Republic of Korea	PT	Portugal		
CU	Cuba	KZ	Kazakstan	RO	Romania		
CZ	Czech Republic	LC	Saint Lucia	RU	Russian Federation		
DE	Germany	LI	Liechtenstein	SD	Sudan		
DK	Denmark	LK	Sri Lanka	SE	Sweden		
EE	Estonia	LR	Liberia	SG	Singapore		

INTERNATIONAL SEARCH REPORT

International Application No
PCT/US 99/28821

A. CLASSIFICATION OF SUBJECT MATTER
IPC 7 G06F9/38 G06F9/46

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)
IPC 7 G06F

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practical, search terms used)

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category *	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
X	US 5 574 939 A (KECKLER STEPHEN W ET AL) 12 November 1996 (1996-11-12)	1,3, 6-12, 14-18, 21,22
Y	column 2, line 1 -column 9, line 11	13
A		19,20, 23-25
X	FILLO M ET AL: "THE M-MACHINE MULTICOMPUTER" PROCEEDINGS OF THE ANNUAL INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE,US,LOS ALAMITOS, IEEE COMP. SOC. PRESS, vol. SYMP. 28, 1995, pages 146-156, XP000585356 ISBN: 0-8186-7349-4	1,3,7,8, 10-12,26
Y	the whole document	9
A		6,13-17, 21,30

-/--

[X] Further documents are listed in the continuation of box C.

[X] Patent family members are listed in annex.

* Special categories of cited documents :

"A" document defining the general state of the art which is not considered to be of particular relevance

"E" earlier document but published on or after the international filing date

"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)

"O" document referring to an oral disclosure, use, exhibition or other means

"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention

"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone

"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art.

"&" document member of the same patent family

Date of the actual completion of the international search	Date of mailing of the international search report
31 May 2000	15/06/2000

Name and mailing address of the ISA	Authorized officer
European Patent Office, P.B. 5818 Patentlaan 2 NL – 2280 HV Rijswijk Tel. (+31-70) 340-2040, Tx. 31 651 epo nl, Fax: (+31-70) 340-3016	Bijn, K

1

C.(Continuation) DOCUMENTS CONSIDERED TO BE RELEVANT

Category °	Citation of document, with indication,where appropriate, of the relevant passages	Relevant to claim No.
X	BEREKOVIC M ET AL: "Hardware realization of a Java virtual machine for high performance multimedia applications" 1997 IEEE WORKSHOP ON SIGNAL PROCESSING SYSTEMS. SIPS 97 DESIGN AND IMPLEMENTATION FORMERLY VLSI SIGNAL PROCESSING (CAT. NO.97TH8262), 1997 IEEE WORKSHOP ON SIGNAL PROCESSING SYSTEMS. SIPS 97 DESIGN AND IMPLEMENTATION FORMERLY VLSI SIGNAL PROCESSING,, pages 479-488, XP002139288 1997, New York, NY, USA, IEEE, USA ISBN: 0-7803-3806-5	1-5,7,8, 10-12, 26-29
Y	the whole document	13
A		14-17
X	US 5 742 782 A (KAMADA EIKI ET AL) 21 April 1998 (1998-04-21)	1,3,6, 26,30
Y	column 5, line 14 -column 6, last line	9
A		8,10-15, 17,20,21
X	KECKLER S W ET AL: "PROCESSOR COUPLING: INTEGRATING COMPILE TIME AND RUNTIME SCHEDULING FOR PARALLELISM" PROCEEDINGS OF THE ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE,US,NEW YORK, IEEE, vol. SYMP. 19, 1992, pages 202-213, XP000325804 ISBN: 0-89791-510-6	1,3,14, 15,17, 21,22,26
A	the whole document	6-13,20, 23,24,30

INTERNATIONAL SEARCH REPORT

Information on patent family members

Inter. nal Application No
PCT/US 99/28821

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
US 5574939 A	12-11-1996	WO 9427216 A	24-11-1994
US 5742782 A	21-04-1998	JP 7281896 A	27-10-1995